Advanced LeetCode/Google Algorithms Cheat Sheet

Time/space complexities included where helpful. Python-first, interview-ready.

Kahn's Algorithm (Topological Sort + Cycle Detection)

```
from collections import deque, defaultdict

def topo_sort_kahn(n, edges):
    # O(V + E) time, O(V + E) space
    adj = defaultdict(list)
    indeg = [0]*n
    for u, v in edges:
        adj[u].append(v); indeg[v] += 1

    q = deque(i for i in range(n) if indeg[i] == 0)
    order = []
    while q:
        u = q.popleft()
        order.append(u)
        for v in adj[u]:
            indeg[v] -= 1
            if indeg[v] == 0:
                q.append(v)

    return order if len(order) == n else []  # empty => cycle
```

Z-Algorithm (Pattern Matching)

```
def z_array(s: str):
    # O(n) time, O(n) space
    n = len(s)
    Z = [0]*n
    l = r = 0
    for i in range(1, n):
        if i <= r:
            Z[i] = min(r - i + 1, Z[i - l])
        while i + Z[i] < n and s[Z[i]] == s[i + Z[i]]:
            Z[i] += 1
        if i + Z[i] - 1 > r:
            l, r = i, i + Z[i] - 1
    Z[0] = n
    return Z

def z_search(text: str, pat: str):
    t = pat + '$' + text
    Z = z_array(t)
    m = len(pat)
    return [i - (m+1) for i in range(m+1, len(t)) if Z[i] == m]
```

KMP (Knuth–Morris–Pratt) String Search

```
def kmp_lps(pat: str):
    # O(m) time
    lps = [0]*len(pat)
    j = 0
    for i in range(1, len(pat)):
        while j and pat[i] != pat[j]:
            j = lps[j-1]
        if pat[i] == pat[j]:
            j += 1
            lps[i] = j
```

```
    return lps

def kmp_search(text: str, pat: str):
    # O(n) time overall
    if not pat: return list(range(len(text)+1))
    lps = kmp_lps(pat)
    i = j = 0
    res = []
    while i < len(text):
        if text[i] == pat[j]:
            i += 1; j += 1
            if j == len(pat):
                res.append(i - j)
                j = lps[j-1]
        else:
            if j: j = lps[j-1]
            else: i += 1
    return res
```

Manacher's Algorithm (Longest Palindromic Substring)

```
def manacher_longest_palindrome(s: str) -> int:
    # O(n) time, O(n) space
    t = '^#' + '#'.join(s) + '#$'
    n = len(t)
    P = [0]*n
    C = R = 0
    for i in range(1, n-1):
        mirror = 2*C - i
        if i < R:
            P[i] = min(R - i, P[mirror])
        while t[i + P[i] + 1] == t[i - P[i] - 1]:
            P[i] += 1
        if i + P[i] > R:
            C, R = i, i + P[i]
    return max(P)
```

Disjoint Set Union (Union-Find)

```
class DSU:
    # near O(α(n)) amortized per op
    def __init__(self, n):
        self.p = list(range(n))
        self.r = [0]*n

    def find(self, x):
        while x != self.p[x]:
            self.p[x] = self.p[self.p[x]]
            x = self.p[x]
        return x

    def union(self, a, b):
        ra, rb = self.find(a), self.find(b)
        if ra == rb: return False
        if self.r[ra] < self.r[rb]:
            self.p[ra] = rb
        elif self.r[ra] > self.r[rb]:
            self.p[rb] = ra
        else:
            self.p[rb] = ra
            self.r[ra] += 1
        return True
```

Dijkstra & 0-1 BFS

```
import heapq
from collections import defaultdict, deque

def dijkstra(n, edges, src):
    # O((V+E) log V)
    adj = defaultdict(list)
    for u, v, w in edges:
        adj[u].append((v, w))
    dist = [float('inf')]*n
    dist[src] = 0
    pq = [(0, src)]
    while pq:
        d, u = heapq.heappop(pq)
        if d != dist[u]: continue
        for v, w in adj[u]:
            nd = d + w
            if nd < dist[v]:
                dist[v] = nd
                heapq.heappush(pq, (nd, v))
    return dist

def zero_one_bfs(n, edges, src):
    # O(V + E) for weights in {0,1}
    adj = defaultdict(list)
    for u, v, w in edges:
        adj[u].append((v, w))
    dist = [float('inf')]*n
    dist[src] = 0
    dq = deque([src])
    while dq:
        u = dq.popleft()
        for v, w in adj[u]:
            if dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
                if w == 0: dq.appendleft(v)
                else: dq.append(v)
    return dist
```

Tarjan's Algorithm (Bridges / Critical Connections)

```
from collections import defaultdict
def bridges(n, edges):
    # O(V + E)
    adj = defaultdict(list)
    for u, v in edges:
        adj[u].append(v); adj[v].append(u)

    disc = [-1]*n
    low  = [-1]*n
    time = 0
    res = []
    def dfs(u, parent):
        nonlocal time
        disc[u] = low[u] = time; time += 1
        for v in adj[u]:
            if disc[v] == -1:
                dfs(v, u)
                low[u] = min(low[u], low[v])
                if low[v] > disc[u]:
                    res.append((u, v))
            elif v != parent:
                low[u] = min(low[u], disc[v])
    for i in range(n):
        if disc[i] == -1:
            dfs(i, -1)
    return res
```

Monotonic Stack (Largest Rectangle in Histogram)

```
def largest_rectangle_area(heights):
    # O(n) time, O(n) space
    heights.append(0)
    st = []
    best = 0
    for i, h in enumerate(heights):
        while st and heights[st[-1]] > h:
            H = heights[st.pop()]
            L = st[-1] if st else -1
            best = max(best, H * (i - L - 1))
        st.append(i)
    heights.pop()
    return best
```

Segment Tree (Range Sum / Point Update)

```
class SegTree:
    # O(log n) per update/query, O(n) space
    def __init__(self, nums):
        n = len(nums); self.n = 1
        while self.n < n: self.n <<= 1
        self.t = [0]*(2*self.n)
        for i, v in enumerate(nums):
            self.t[self.n + i] = v
        for i in range(self.n-1, 0, -1):
            self.t[i] = self.t[i<<1] + self.t[i<<1|1]

    def update(self, i, val):
        i += self.n
        self.t[i] = val
        i >>= 1
        while i:
            self.t[i] = self.t[i<<1] + self.t[i<<1|1]
            i >>= 1

    def query(self, l, r):  # [l, r)
        res = 0
        l += self.n; r += self.n
        while l < r:
            if l & 1: res += self.t[l]; l += 1
            if r & 1: r -= 1; res += self.t[r]
            l >>= 1; r >>= 1
        return res
```

Fenwick Tree / Binary Indexed Tree (Prefix Sums)

```
class Fenwick:
    # O(log n) per op, O(n) space
    def __init__(self, n):
        self.n = n
        self.bit = [0]*(n+1)

    def add(self, i, delta):  # 0-indexed
        i += 1
        while i <= self.n:
            self.bit[i] += delta
            i += i & -i

    def sum(self, i):  # prefix sum [0..i], 0-indexed
        i += 1
        s = 0
        while i > 0:
            s += self.bit[i]
            i -= i & -i
        return s
```

```
    def range_sum(self, l, r):
        return self.sum(r) - self.sum(l-1)
```

Trie (Prefix Tree)

```
class TrieNode:
    __slots__ = ("children", "end")
    def __init__(self):
        self.children = {}
        self.end = False

class Trie:
    # insert/search/prefix in O(L), space O(total chars)
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word: str) -> None:
        cur = self.root
        for ch in word:
            if ch not in cur.children:
                cur.children[ch] = TrieNode()
            cur = cur.children[ch]
        cur.end = True

    def search(self, word: str) -> bool:
        cur = self.root
        for ch in word:
            if ch not in cur.children:
                return False
            cur = cur.children[ch]
        return cur.end

    def startsWith(self, prefix: str) -> bool:
        cur = self.root
        for ch in prefix:
            if ch not in cur.children:
                return False
            cur = cur.children[ch]
        return True
```